

**APPLICATION FOR UNITED STATES LETTERS PATENT FOR**  
**A**  
**METHOD, APPARATUS & COMPUTER PROGRAM PRODUCT FOR**  
**BORROWED-VIRTUAL-TIME SCHEDULING**

The assignee is Cisco Technology, Inc., a California corporation having an office at  
170 West Tasman Drive, San Jose CA 95134.

<u><b>INVENTORS</b></u>	<u><b>Citizenship</b></u>	<u><b>Residence City and State</b></u>
Duda, Kenneth J.	USA	2445 Summit Drive, Hillsborough, CA 94010
Cheriton, David R.	Canada	131 Cowper St., Palo Alto, CA 94301

**A**

**METHOD, APPARATUS & COMPUTER PROGRAM PRODUCT FOR  
BORROWED-VIRTUAL-TIME SCHEDULING**

**Background of the Invention**

**5    *Field of the Invention***

This invention relates to the field of scheduling electronic and computer resources.

***Background***

There are many circumstances where a resource is shared between elements. For example, the resources of a computer system are shared by the programs executing in the computer system. In a computer system, one particular example of a shared resource is the time available to the processor that executes programs stored in the computer's memory. This resource (the processor time) is allocated to elements in the computer system (threads-of-execution) that use the processor time.

Another example of a shared resource occurs in an output-queued data switch. Such a switch dispatches data received from an input port to a queue associated with the destination output port. The queue then feeds the output port. There can be a number of queues for each output port. The shared resource is the amount of time each queue (an element) has access to the output port. In other words, the shared resource is the output port's bandwidth.

In both of these cases (and many others), the available time of the resource is shared between multiple elements under control of a scheduling mechanism that implements a scheduling algorithm. The scheduling mechanism can include programmed processes that are executed by a processor to effect a scheduling algorithm, circuitry that effectuates a scheduling algorithm, or other known mechanisms. In addition, the scheduling mechanism may be preemptive or non-preemptive. In the case of a computer system, a preemptively

scheduling operating system allocates a quantum of processor time to an element and potentially schedules another element (for example, when the quantum of time expires, if the element relinquishes the processor, or for other reasons). In the case of a non-preemptively scheduling operating system, each element must explicitly give other elements the opportunity to access the processor.

There are many known scheduling algorithms. Some of these algorithms are designed to fairly allocate the resource between the elements. Fairness can be defined as each element asymptotically (over time) receiving its weighted share of the resource's time. Other definitions of fairness are known in the art.

The greater the element's weight, the greater its fair share of the resource. Letting  $R(t)$  denote the set of all competing elements at time  $t$ , the instantaneous fair share of an element  $i$  at time  $t$  is defined as:

$$f_i(t) = \frac{w_i}{\sum_{j \in R(t)} w_j}$$

Known scheduling algorithms include weighted fair queuing scheduling, round robin scheduling, priority based scheduling, deadline scheduling, and other scheduling algorithms. One problem with the known art is that often an element is sensitive to latency. One example of a latency-sensitive element in a computer system is a program that displays video images. Each frame needs to be promptly displayed after the expiration of a timer. If the frame is delayed, the displayed video will appear jerky. Another example of a latency-sensitive element is a high service requirement data packet queue (containing, for example, voice or video data) flowing through a data switch. The data switch performance will be inadequate if the data packet is not transferred to the output port of the data switch within its service requirement. In both of these examples, the purpose of the element is degraded if the required resource is not available when needed.

Deadline scheduling techniques allow an element to declare its future resource needs and to request that the scheduler accommodate the declared need. The scheduler may accept

or decline this request (if declined, the scheduler treats the requesting element in its default manner). One problem with this approach is that it is difficult for an element to determine how much of the resource the element requires. Other problems are: that if multiple elements declare more resource capability than is available, other elements may be starved for the resource; that the system is unpredictable if some of the declaring elements that request resource are accepted and other elements denied; that the elements often incorrectly estimate their resource need; and that the deadline scheduling is complex to implement.

Although, latency-sensitive elements can be scheduled using a priority-based algorithm such scheduling is often unfair and can lead to resource starvation of lower priority elements.

It would be advantageous to schedule a resource between elements to maintain a fair long-term allocation of the resource to the elements while still satisfying the responsive needs of latency-sensitive elements. Such scheduling would improve the performance of a device that maintains a fair long-term allocation of a resource to multiple elements while still handling latency-sensitive elements. As applied to a computer system, such a thread-of-execution scheduling system would improve the performance of the computer system. As applied to a data switch, such an output port scheduling algorithm would improve the performance of the latency-sensitive data streams through the switch.

### **Summary of the Invention**

The present invention includes a scheduling mechanism that fairly allocates a resource to a number of schedulable elements of which some are latency-sensitive. The invention tracks each element's use of the resource by determining the element's virtual time. An active element is selected from the elements that are ready to use the resource by determining the element that has the smallest effective virtual time. The effective virtual time is the element's actual virtual time modified by a borrowed virtual time value. When an element has a short-term need for the resource, it can borrow the privilege to run by borrowing virtual time. As the element uses the resource, it consumes virtual time according to its weight.

When the elements are scheduled for the resource, the ready element having the smallest virtual time is selected. The invention enforces long-term fairness to each element while allowing latency-sensitive elements to be preferably selected. One preferred embodiment is a thread-of-execution scheduler for a computer system. This embodiment allocates the CPU time (the resource) between multiple threads (the elements). Latency-sensitive threads (for example, video display programs) can borrow virtual time so that they are more likely to be scheduled when a condition is satisfied (for example, expiration of a timer that triggers the display of the next video frame). Another preferred embodiment is a queue scheduler for an output-queued data switch. Here multiple queues feed an output port. The multiple queues have different service requirements. The invention provides long-term fairness while still satisfying queues that contain latency-sensitive data.

### **Description of the Drawings**

- Fig. 1 illustrates a computer system capable of using the invention in accordance with a preferred embodiment;
- Fig. 2A illustrates a data switch in accordance with a preferred embodiment;
- Fig. 2B illustrates a switch fabric for the data switch shown in Fig. 2A;
- Fig. 3 illustrates a scheduling process in accordance with a preferred embodiment;
- Fig. 4 illustrates a process to update the virtual time of an element in accordance with a preferred embodiment;
- Fig. 5 illustrates a process to select an element in accordance with a preferred embodiment using a computer system;
- Fig. 6A illustrates a process to update the system reference-use in accordance with a preferred embodiment;

- Fig. 6B illustrates a process to update the system reference-use in accordance with another preferred embodiment;
- Fig. 7 illustrates a process to update the virtual time of a previously inactive element in accordance with a preferred embodiment;
- 5 Fig. 8A illustrates a record virtual time process performed when an element goes inactive in accordance with a preferred embodiment;
- Fig. 8B illustrates a ready inactive element in accordance with a preferred embodiment;
- Fig. 9 illustrates a process to create a new element in accordance with a preferred embodiment;
- 10 Fig. 10 illustrates how a process can borrow and repay virtual time to handle latency-sensitive conditions in accordance with a preferred embodiment; and
- Fig. 11 illustrates a BVT scheduling process within a data switch in accordance with a preferred embodiment.
- 15

## **Description of the Preferred Embodiments**

### ***Notations and Nomenclature***

The following 'notations and nomenclature' are provided to assist in the understanding of the present invention and the preferred embodiments thereof.

20 **Thread, Thread-of-execution** — A thread-of-execution is a sequence of control within a programmed-process. A traditional single-threaded programmed-process follows a single sequence of control while executing. A multithreaded programmed-process has several sequences of control, and is capable of several independent actions.

**Pointer** — A pointer is a data value that is used to reference a data structure or an object. One skilled in the art will understand that “pointer” includes, without limitation, a memory address to, or a value used to calculate the address to the information of interest and any functional equivalents including handles and similar constructs.

5        **Procedure** — A procedure can be a self-consistent sequence of computerized steps that lead to a desired result. These steps are defined by one or more computer instructions. These steps are performed by a computer executing the instructions that define the steps. Thus, the term “procedure” can refer to a sequence of instructions, a sequence of instructions organized within a programmed-procedure or programmed-function, or a sequence of  
10 instructions organized within programmed-processes executing in one or more computers. A procedure also includes operations performed by specially designed hardware logic that lead to a desired result.

15        **Process** — A process is a sequence of method steps as compared to the computer science term that associates memory with a thread of execution (that is, the term “process” as used herein does not refer to programmed-processes serviced by a computer’s operating system).

### ***Detailed Description***

20        The invention can be incorporated within at least two preferred embodiments. One preferred embodiment is that of a scheduler for threads-of-execution within a computer system. Another preferred embodiment is that of a scheduler for different output queues for an output queued data switch. The invention schedules elements (threads, queues) to use a resource (CPU time, output port bandwidth) such that each element receives its fair share of the resource over time while still being responsive to short-term latency-sensitive needs.

25        **Fig. 1** illustrates a computer, indicated by general reference character **100**, that incorporates the invention. The computer **100** includes a processor **101** that incorporates a central processor unit (CPU) **103**, a memory section **105** and an input/output (I/O) section **107**. The I/O section **107** is connected to a user interaction mechanism **111**, a disk storage

unit **113** and a CD-ROM drive unit **115**. The CD-ROM drive unit **115** can read a CD-ROM medium **117** that typically contains a program and data **119**. The CD-ROM drive unit **115** (along with the CD-ROM medium **117**) and the disk storage unit **113** comprise a filestorage mechanism. Some embodiments of the invention include a network interface **121** that connects the computer **100** to a network **123**. A program **125** incorporating the invention can be stored in the memory section **105**. The instructions contained in the program **125** are executed by the CPU **103**. One skilled in the art will understand that not all of the displayed features of the computer **100** need to be present for the invention. In addition, any removable media can be used in place of the CD-ROM medium **117** so long as a suitable device is available to read the media. The program **125** can be loaded into memory from the file system or from the network **123**.

The time spent by the CPU **103** executing code is a resource. One scheduling issue is how to share the available CPU time between threads-of-execution. Generally, the operating system in the computer contains a scheduler that allocates CPU time to threads-of-execution. Each thread is available to use the CPU (that is, the thread is ready) unless the thread is blocked (not ready to be activated). The active element is the one that is to use the resource. Thus, the active thread is the thread that is to use the CPU.

One problem with scheduling threads in a computer system is that it is difficult to fairly accommodate both latency-sensitive threads and latency-insensitive threads. In one preferred embodiment the borrowed-virtual-time (BVT) scheduler of the invention provides a thread scheduler that is both fair and accommodates latency-sensitive threads.

In another preferred embodiment, the BVT scheduler can be used with a data switch to allocate the bandwidth of an output port (by allocating the time that the output port is available to a queue). The BVT scheduler provides fair use of the output port bandwidth while providing lower latency service for latency-sensitive traffic for data having different latency-sensitivity characteristics.



Fig. 2A illustrates a data switch, indicated by general reference character **200**, showing some of the components that are included in a data switch. The data switch **200** includes a switch fabric **201**, a first input port **203**, an nth input port **205**, a first output port **207**, and an nth output port **209**. The switch fabric **201** provides connections between any of the input ports and any of the output ports. A data packet or cell (not shown) enters the data switch **200** at one of input ports **207**, **209**. The data switch **200** examines the contents of the data packet (or cell) and determines the service requirements and destination output port or ports required by the data packet. The data packet (or pointer to the data packet) is then queued into one of the queues that feed the specified output port. The queue selection can be based on the data packet's service requirements (for example, but without limitation, latency-sensitive data packets can be placed in a different queue than latency-insensitive data packets, or multicast packets).

Hereinafter the term "data packet" refers to both a data cell (a fixed length unit of data) and a data packet (a variable length unit of data).

Fig. 2B details an output queued switch fabric, indicated by general reference character **250**, that supports an output-queued capability. Once the data packet enters the data switch **200**, the data packet is stored in a memory **251**. The switch fabric **250** also includes a packet dispatch mechanism **253** and a BVT scheduling mechanism **255**. The packet dispatch mechanism **253** examines the contents of the data packet, and determines the level of service and the output port(s) required by the data packet. Depending on the level of service required (for example, latency-sensitive or latency insensitive) and/or type of service required (for example, multicast or unicast), a pointer to the data packet is stored (for example, but without limitation) in either a 'bulk service' queue **259**, an 'interactive latency service' queue **261**, or a 'real-time latency service' queue **263** for the specified output port. The BVT scheduling mechanism **255** then fairly allocates the bandwidth of the output port to each of these queues while satisfying latency-sensitive packet requirements as is subsequently described. The number of queues feeding the output port need not be three. BVT scheduling can be

accomplished as long as there are two or more queues and an algorithm for determining which queue to use for any given data packet.

One skilled in the art will understand that the bandwidth is a function of the speed of the output port and the time required to send information through the port. Thus, allocation of the time the output port is available to a queue is equivalent to allocating bandwidth to the queue.

Although much of the subsequent description of the invention is directed toward the scheduling of a computer's CPU, one skilled in the art will understand how to use BVT scheduling in an output-queued data switch such as the one shown in Fig. 2A and Fig. 2B.

**Fig. 3** illustrates a preemptive scheduling process, indicated by general reference character **300**, used to schedule a resource among a plurality of elements. In the case of a preemptive scheduling algorithm performed by a computer that is executing procedures, the preemptive scheduling process **300** is often initiated as a result of a system timer interrupt, when a thread yields, or when a thread blocks or unblocks. One skilled in the art will understand that other mechanisms can be used within the computer context and that yet other mechanisms can be used within a data switch to effectuate the preemptive scheduling process **300**. When interrupt driven, the preemptive scheduling process **300** initiates when a timer interrupt occurs at an 'enter timing interrupt' terminal **301** and continues to a 'prior art timer interrupt' procedure **303**. The 'prior art timer interrupt' procedure **303** performs relevant prior art processing for the timer interrupt. Then the preemptive scheduling process **300** continues to a 'decrement quantum counter' procedure **305** that decrements the quantum counter for the active element.

Next, a 'quantum complete' decision procedure **307** determines whether the quantum allocated to the element has completed. If the quantum has not expired, the preemptive scheduling process **300** exits through a 'return from interrupt' terminal **309**. The quantum is set to run for some multiple of clock ticks and is the period-of-use of the resource by the

element. That is, an element is allowed to have exclusive access to the resource for some number of clock ticks (the quantum or scheduled period-of-use).

Thus, for a thread scheduling BVT embodiment, the 'quantum complete' decision procedure **307** detects the expiration of a period-of-use for the resource (the CPU) that is allocated to the current element (a thread). Thus, the preemptive scheduling process **300** detects the expiration a period-of-use of the resource for the currently scheduled element. For a data switch, the period-of-use is the time required to send a packet or cell of data through the output port.

Once the element's quantum expires (because sufficient clock ticks have occurred to satisfy the 'quantum complete' decision procedure **307**), the preemptive scheduling process **300** continues to a 'reset quantum counter' procedure **311**. The 'reset quantum counter' procedure **311** resets the element's quantum counter for the element's next scheduled period-of-use.

A data switch does not require the determination of a quantum. The amount of data that is sent through the output port can be used to instantaneously determine the use of the output port's bandwidth as is subsequently described with respect to Fig. 6B.

An 'update virtual time' procedure **313** then updates the element's virtual time by adding a resource cost to the virtual time maintained for the element as subsequently described with respect to Fig. 4. The resource cost depends on the assigned weight of the element and the amount of time the element had access to the resource. The virtual time for the element also can be updated when the element becomes ready to use the resource as is subsequently described with respect to Fig. 7. Once the virtual time for each ready element has been determined, a 'select active element' procedure **317** selects and activates the active element from the collection of ready elements. The 'select active element' procedure **317** is subsequently described with respect to Fig. 5. It includes procedures for selecting the ready element that has the smallest effective virtual time. In addition, the preemptive scheduling process **300** updates the system's virtual time (a system reference-use for the resource) at an

‘update system virtual time’ procedure **319** (subsequently described with respect to Fig. 6A and Fig. 6B). Finally, the preemptive scheduling process **300** completes through the ‘return from interrupt’ terminal **309** allowing the activated thread to use the resource (CPU).

For a data switch the resource cost can be the size of the packet divided by the weight  
5 of the queue feeding the output port.

Other mechanisms known in the art can be used to determine the completion of a quantum other than by decrementing a counter.

The quantum counter for the thread scheduler, and a data packet completion detector in a data switch both serve to detect the expiration of a period-of-use of the resource (CPU  
10 time, and output port bandwidth) allocated to the element (thread or output port queue) utilizing the resource.

The ‘update virtual time’ procedure **313** updates a measure-of-use of the resource for the currently used element. The measure-of-use is a function of the period-of-use and a measure-of-use adjustment (such as a thread’s quantum of execution and the weight of the  
15 thread).

The ‘select active element’ procedure **317** selects one of the elements to use the resource for a subsequent quantum of time or for the time required to transfer a subsequent data packet through the output port. This selection is responsive to the measure-of-use for each ready element. In a preferred embodiment, the element that has the smallest effective  
20 virtual time (EVT) is activated. Each element’s EVT is responsive to the element’s actual virtual time (AVT) minus an element-specific selection adjustment (the virtual time borrowed by the element). This is described in more detail with respect to Fig. 5.

One skilled in the art will understand how to apply the techniques described herein to non-preemptive scheduler mechanisms such that the period of use need not be scheduled, but  
25 can result from the programmed-processes relinquishing the CPU.

As was previously mentioned, the preemptive scheduling process **300** shown in Fig. 3 is directed toward a preferred embodiment of applying BVT scheduling to scheduling the access of multiple threads to CPU time.

In another preferred embodiment, for scheduling the access of multiple queues to an output port in a data switch, the processes related to determining the quantum of time used by the element is not needed as the data switch can directly determine the amount of bandwidth consumed by the transfer of the data packet. In addition, the data switch need not use interrupts because the functionality required to determine the resource utilization can be directly accomplished by circuitry. In this embodiment, the measure-of-use adjustment is dependent on the service requirement for the queue (for example and without limitation, interactive, batch, real-time).

**Fig. 4** illustrates an update virtual time process, indicated by general reference character **400**, that is invoked by the 'update virtual time' procedure **313** and that initiates at a 'start' terminal **401**. The update virtual time process **400** updates the element's virtual time by adding a resource cost to the element by an 'add resource cost' procedure **403**. When the resource is the CPU time used to execute a thread, the resource cost is the CPU cost. The CPU cost can be determined by dividing the quantum by the weight assigned to the thread. The weight assigned to the thread indicates the thread's long-term proportional share of the CPU. Thus, in a preferred embodiment the AVT of the element's use of the resource (the measure-of-use for the resource) is increased as:

$$A_i \leftarrow A_i + \frac{q}{w_i}$$

where the AVT for element  $i$  is given by  $A_i$  and the CPU cost is the quantum divided by the weight for the current element. Thus, when the element does not utilize the resource, the AVT for the element remains constant.

In the case of a data switch, the resource cost can be the weighted amount of the bandwidth that was used by the transfer of a data packet or data cell through the output port.

This can be determined by the size of the packet divided by the weight assigned to the packet's queue.

The update virtual time process **400** continues to a 'maximum virtual time' decision procedure **405** to determine whether the AVT value has reached a limit. The purpose of the update virtual time process **400** is to normalize the virtual time values of all the elements and of the system when the virtual time value for a particular element overflows. Thus, this process has no direct affect on the scheduling decision (but does solve the infrequent problem of virtual time value overflow when practicing the invention). If the AVT value has not reached a limit, the update virtual time process **400** completes through an 'end' terminal **407**.

However, if the AVT value has exceeded the limit, the update virtual time process **400** continues to an 'adjust system virtual time' procedure **409** that adjusts the system's virtual time (SVT) by subtracting a normalization constant from the SVT. The SVT is further described with respect to Fig. 6A and Fig. 6B. The update virtual time process **400** then iterates each element at an 'iterate elements' procedure **411**. When all of the elements have been iterated, the update virtual time process **400** completes through the 'end' terminal **407**. The AVT for each iterated element is also reduced by the normalization constant (while staying positive) by an 'adjust virtual time of element' procedure **413**. Thus, when an element's AVT passes a limit, the system's virtual time and every element's AVT is reduced by a normalization constant.

**Fig. 5** illustrates a 'select element' process, indicated by general reference character **500**, that is invoked by the 'select active element' procedure **317** of Fig. 3. The 'select element' process **500** initiates at a 'start' terminal **501** and continues to a 'set max effective virtual time' procedure **503** that initializes the TEVT variable to the maximum possible effective virtual time (EVT). Next, an 'initialize pointer variable' procedure **505** sets pointer variable RunEle to Null. An 'iterate ready elements' procedure **507** then iterates over each ready element (one that is ready to consume the resource) to locate the element that has the smallest EVT. When the ready elements have been iterated, a 'return active element pointer' procedure **509** returns the pointer and the 'select element' process **500** completes through an

‘end’ terminal **511**. As each ready element is iterated, a ‘smallest effective virtual time’ decision procedure **513** determines whether the iterated element’s EVT is smaller than the value contained in TEVT. If not, the ‘select element’ process **500** continues to the ‘iterate ready elements’ procedure **507** to iterate the next ready element. Otherwise, the ‘select element’ process **500** continues to a ‘set new minimum effective virtual time’ procedure **515** that stores the iterated element’s EVT in TEVT. Next, a ‘set selected element pointer’ procedure **517** stores the pointer to the element in RunEle and the ‘select element’ process **500** continues to the ‘iterate ready elements’ procedure **507**. Thus, the element with the smallest effective virtual time (EVT) is the one selected to next receive access to the resource.

The EVT ( $E$ ) for element  $i$  is defined to be:

$$E_i = A_i - b_i$$

where  $b_i$  is the amount of virtual time borrowed by the  $i^{th}$  element (the borrowed virtual time), and  $A_i$  is the actual virtual time as has been previously discussed.

Although described in terms of steps performed by a computer, one skilled in the art will understand that the above algorithm can be implemented by circuitry. Thus, in an output queued data switch, the steps used to decide which queue is to be selected to use the output port is implemented using circuitry.

**Fig. 6A** illustrates an ‘update system virtual time’ process, indicated by general reference character **600**, that is invoked by the ‘update system virtual time’ procedure **319** of Fig. 3 and that initiates at a ‘start’ terminal **601**. The ‘update system virtual time’ process **600** continues to an ‘update system virtual time’ procedure **603** and completes through an ‘end’ terminal **605**. The ‘update system virtual time’ procedure **603** updates the system’s VT as:

$$\text{System.VT} = \max(\text{System.VT}, \min(\text{System.VT} + R + \text{RCost}, \text{EVT}));$$

where RCost is the resource cost of the element that consumed the resource; R is the system virtual time convergence rate; and EVT is the effective virtual time of the newly selected element. Thus, if there is no active element the system’s VT increases by R. The system

virtual time represents the general area of virtual time where most of the element's virtual time lies. The system virtual time convergence rate (R) can be viewed as the rate an indebted element's debt is forgiven when no other elements want the resource. For a computer operating system, the system virtual time is the system reference-use for the processor  
5 executing the thread.

**Fig. 6B** illustrates an 'update system virtual time' process, indicated by general reference character **610**. This process can also be used to update the system's virtual time if the process described with respect to Fig. 8A is used to save information related to the time an element was blocked. The 'update system virtual time' process **610** initiates at a 'start' terminal **611** and is invoked by the 'update system virtual time' procedure **319** of Fig. 3. The  
10 'update system virtual time' process **610** continues to a 'determine maxchange' procedure **613** that determines a MaxChange value that is the resource cost of the active element inflated by a taxrate value. A 'determine meanAVT' procedure **615** averages the AVT values over all of the elements that are ready to run to determine the mean of the actual virtual time (a virtual time average) for all elements currently capable of using the resource. An 'update system  
15 virtual time' procedure **617** updates the system's virtual time. The 'update system virtual time' process **610** completes through an 'end' terminal **619**.

The 'update system virtual time' procedure **617** updates the system's virtual time as:

System.VT= += max(-MaxChange,  
20 min(MaxChange, MeanAVT-System.VT));

Thus, System.VT (SVT) converges towards the mean actual virtual time for each ready element; and System.VT never changes faster than MaxChange. The 'update system virtual time' process **610** converges towards the mean AVT of the ready elements. The System.VT value represents mean of the virtual time of the ready elements. Once this  
25 reference is determined, the scheduler can determine the AVT for a blocked element when it is made ready, as will be subsequently described with respect to Fig. 7 and Fig. 8B. In addition, by not changing the System.VT value by more than the inflated resource cost of



the just active element, the scheduler can forgive virtual time debts by the taxable amount, or tax elements that have a saving. In addition:

$\max(-\text{MaxChange}, \min(\text{MaxChange}, \text{MeanAVT} - \text{System.VT}))$ ;

- 5 represents an adjustment value that is used to adjust the system reference-use (the system virtual time). The adjustment value can be used to adjust the AVT of the blocked elements so that the AVT of the blocked elements substantially tracks or converges to the SVT.

For a data switch, the system reference-use is similarly determined. For example, after each packet is transferred from the queue to the output port(s), the measure-of-use is  
10 determined from the packet size and the queue's weight (for example, the packet size divided by queue weight) and this is defined to be the resource cost. The system also determines the MeanAVT for each non-empty queue. Using these definitions, the system reference-use is determined as was previously described.

**Fig. 7** illustrates an 'update virtual time for unblocked element' process, indicated by  
15 general reference character **700**, that can be invoked when the condition blocking the execution of the thread is satisfied (for example, occurrence of a timer completion, a device interrupt, or some other exception or condition). The process **700** initiates at a 'start' terminal **701** and continues to a 'set element's virtual time' procedure **703** that updates the AVT for the element according to:

20  $\text{Ele.AVT} = \max(\text{Ele.AVT}, \text{System.VT} - \text{Lag})$

where Lag is a constant such as the CPU cost of for executing a typical element for a quantum.

Once the element's AVT is updated, the element is made ready so that it can compete for the resource by a 'ready' procedure **705**. The process **700** completes through an 'end'  
25 terminal **707**.

The purpose of the process **700** is to compensate for elements that have been blocked for significant amounts of time. Because the blocked elements are not executing, their AVT

is not increased (in some embodiments) during the time blocked. Thus, the blocked element's AVT remains low as compared to the elements that are consuming resource. Without this adjustment, the newly unblocked element would dominate the other elements for some period of time.

5           The Lag value is used so that elements that are briefly suspended are not penalized in that they are more likely to run than elements that have not been blocked (thus, a thread blocked by a page fault is more likely to run).

10           **Fig. 8A** illustrates a record virtual time process, indicated by general reference character **800**, that records information about the element's scheduling state at the time the element blocked. The record virtual time process **800** initiates at a 'start' terminal **801** when the element becomes blocked. The record virtual time process **800** continues to a 'save sleep real time' procedure **803** that stores the real time that the element blocked. A 'save sleep system virtual time' procedure **805** saves the system virtual time. Both of these values are associated with the element. Thus, each blocked element is associated with its own respective values. The record virtual time process **800** completes through an 'end' terminal **807**.

15           Thus, when the element becomes blocked, the system captures information that can be used to reevaluate the element's AVT when the element subsequently becomes ready.

20           **Fig. 8B** illustrates a ready blocked element process, indicated by general reference character **810**, that can be used when an element, that has information stored by the record virtual time process **800** of Fig. 8A, becomes ready. The ready blocked element process **810** initiates at a 'start' terminal **811** and advances to a 'was element in debt' decision procedure **813**. The 'was element in debt' decision procedure **813** determines whether the element was in debt at the time the element blocked (that is, at the time the element blocked, whether it had received more than its fair share of the resource—possibly by borrowing). If so, the ready blocked element process **810** continues to an 'is element in debt' decision procedure **815** that determines whether the element is currently in debt (whether AVT is greater than the current

25

SVT). If so, the ready blocked element process **810** does not adjust the element's AVT and the ready blocked element process **810** completes through an 'end' terminal **817**.

However, if the 'is element in debt' decision procedure **815** determines that the element is no longer in debt (that is, the SVT has passed the element's AVT), then a 'reset debt' procedure **819** sets the element's AVT to the SVT (thus, the element continues to consume resource in rough parity with the other ready elements so as to maintain fairness when the element becomes ready). Then, the ready blocked element process **810** completes through the 'end' terminal **817**.

Looking again at the 'was element in debt' decision procedure **813**, if the element was not in debt when it became blocked (that is, that the element had not received its fair share of the resource at the time it blocked), the ready blocked element process **810** continues to a 'determine intermediate values' procedure **821**. The 'determine intermediate values' procedure **821** evaluates:

```
sav = Ele.SSVT-Ele.AVT;  
st = RT-Ele.SRT;  
tax = sav*(st*taxrate);
```

where `sav` stores the amount of virtual time used by the element less the element's fair share at the time the element blocked—the savings; `st` stores the amount of time the element was blocked—the sleep time; `taxrate` stores the rate that the debt is forgiven or the savings taxed; and `tax` stores the amount of time that will be removed from the savings.

An 'element retain some savings' decision procedure **823** determines whether the element is to retain some of its savings when it becomes ready. If not, a 'forfeit savings' procedure **825** sets the element's AVT to the SVT (thus, the element continues to consume resource in rough parity with the other ready elements so as to maintain fairness when the element becomes ready). Then, the ready blocked element process **810** completes through the 'end' terminal **817**.

However, if the 'element retain some savings' decision procedure **823** determines that the element is to retain some of its savings, the ready blocked element process **810** continues to a 'retain some savings' procedure **827** that sets the element's AVT to the current SVT less the taxed savings. Then, the ready blocked element process **810** completes through the 'end' terminal **817**.

Thus, when the element becomes unblocked, its AVT is adjusted so that the unblocked element fairly competes with other ready elements and possibly reinstating some savings or debt.

Yet another preferred embodiment adjusts each blocked element's AVT dependent on the changes to the SVT. Thus, when the SVT is modified (for example, by the 'update system virtual time' process **610** of Fig. 6B) the AVT of each blocked element is similarly modified so that the blocked elements continue to substantially track the changes in the SVT.

**Fig. 9** illustrates a 'create new element' process, indicated by general reference character **900**, that is used in a computer that has the capability of creating new threads or in a data switch that has dynamically created queues. The following description is directed toward a computer system where the new element is a new thread. Similar operations can be performed by the data switch. In a computer system, the new element (the child element) is generated by an existing element (the parent element).

The 'create new element' process **900** initiates at a 'start' terminal **901** and continues to an 'allocate element storage' procedure **903** that allocates storage for the new element. Once storage is allocated for the new element, an 'initialize element' procedure **905** initializes the new element. Next, a 'set virtual time' procedure **907** sets the initial AVT for the child element so that the child element will fairly share the resource with respect to its weight. Next, the 'create new element' process **900** continues to a 'ready element' procedure **909** that conditions the element to compete for the resource and makes the element such that it can be selected by the 'select element' process **500** of Fig. 5. Finally, the 'create new element' process **900** completes through an 'end' terminal **911**.

In one preferred embodiment, the 'set virtual time' procedure **907** copies the AVT from the parent element to the child element. This approach causes the child element to have the same advantage or penalty with respect to the other elements as does the parent element. In another preferred embodiment, the 'set virtual time' procedure **907** copies the SVT to the child element's AVT. This embodiment starts the child element without an advantage or penalty with respect to mean of the other ready elements.

**Fig. 10** illustrates a virtual time borrowing process, indicated by general reference character **1000**, used by a thread. The virtual time borrowing process **1000** initiates at a 'start' terminal **1001** prefatory to executing latency-sensitive code. The virtual time borrowing process **1000** then continues to a 'borrow virtual time' procedure **1003** that specifies the amount of virtual time to borrow to the value of  $b$ . Next, the virtual time borrowing process **1000** continues to a 'block on condition' procedure **1004** that stops execution of the thread that borrowed the time. Thus, when the thread becomes ready, the 'select element' process **500** will be more likely to select the thread as compared to other threads that have not borrowed virtual time because the EVT for an element is the element's AVT minus  $b$  (recall that the EVT is used schedule a thread). After borrowing the virtual time, a 'latency-sensitive' procedure **1005** will more likely be scheduled. After the 'latency-sensitive' procedure **1005** completes, the virtual time borrowing process **1000** can optionally return the borrowed virtual time at a 'return borrowed virtual time' procedure **1007** (which specifies the amount of virtual time to borrow to the value of zero) and compete through an 'end' terminal **1009**. This particular embodiment is useful when the thread includes both latency-sensitive code and latency-insensitive code. In particular, the virtual time borrowing process **1000** would be invoked just before blocking for a condition that, when complete, requires a rapid response. The value of  $b$  can also be specified external to the element. For example,  $b$  can be a task parameter that can be adjusted by a user or system administrator using appropriate configuration programs, techniques or methods.

**Fig. 11** illustrates a BVT process, indicated by general reference character **1100**, for selecting a queue to feed an output port of a data switch. The BVT process **1100** initiates at a

‘start’ terminal **1101** and continues to an ‘assign Q weights’ process **1103**. The ‘assign Q weights’ process **1103** locates each queue and specifies a weight for each queue. The weight again specifies the element’s long-term share of the resource with respect to the other elements competing for the resource. Next, an ‘assign b values to Qs’ process **1105** assigns to each queue the borrowed virtual time parameter based on the latency-sensitivity of the data traffic assigned to the queue. The ‘assign Q weights’ process **1103** and the ‘assign b values to Qs’ process **1105** can be performed during the initialization phase of the data switch. In alternative embodiments, these processes can be configured manually or automatically while the data switch operates.

Next, a ‘select Q’ process **1107** determines which queue serving the output port has the smallest EVT. A data packet (pointed to by the first item in the selected queue) is then transferred from the memory to the output port by a ‘transfer data in Q’ process **1109**. Next, a ‘track virtual time’ process **1111** updates the AVT for the queue. The BVT process **1100** continues to the ‘select Q’ process **1107**.

One skilled in the art will understand that the invention provides fair scheduling while still allowing latency-sensitive elements (such as latency-sensitive threads and queues) to receive responsive scheduling.

From the foregoing, it will be appreciated that the invention has (without limitation) the following advantages:

- 1) the invention maintains long-term fairness for scheduled elements that use a resource while accommodating short-term latency-sensitive needs of the scheduled elements;
- 2) the invention provides better response times for latency sensitive elements;
- 3) the invention is less complex to implement than deadline scheduling approaches and provides a more fair allocation of the resource to multiple competing elements; and
- 4) the elements need not perform complex determinations as to how much resource that they need to preallocate.

Although the present invention has been described in terms of the presently preferred embodiments, one skilled in the art will understand that various modifications and alterations may be made without departing from the scope of the invention. Accordingly, the scope of the invention is not to be limited to the particular invention embodiments discussed herein.

1512220" 9032260